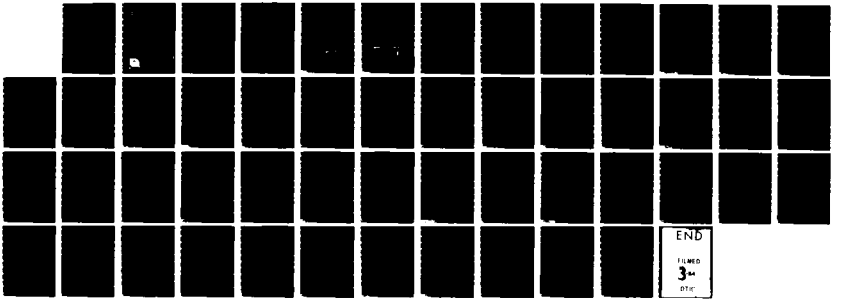
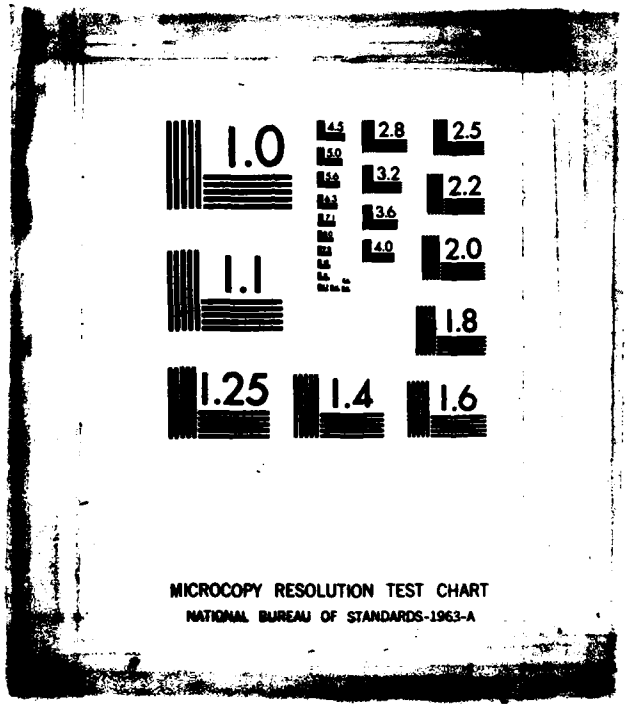
AD-A137 335 OPERATIONAL SOFTWARE TECHNOLOGY WORKING GROUP REPORT (IDA/OSD R&M (INSTIT.. (U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA SCIENCE AND TECH.. L E DRUFFEL 1/1

UNCLASSIFIED AUG 83 IDA-D-38 IDA/HQ-83-25900 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

IDA RECORD DOCUMENT D-38

OPERATIONAL SOFTWARE TECHNOLOGY WORKING GROUP REPORT

(IDA/OSD R&M STUDY)

Lt. Col. Lawrence E. Druffel, USAF
Working Group Chairman

August 1983

The views expressed within this document are those of the working group only. Publication of this document does not indicate endorsement by IDA, its staff, or its sponsoring agencies.

Prepared for
Office of the Under Secretary of Defense for Research and Engineering
and
Office of the Assistant Secretary of Defense
(Manpower, Reserve Affairs and Logistics)

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION

IDA Log No. HQ 83-25900

The work reported in this document was conducted under contract MDA 903 79 C 0018 for the Department of Defense. The publication of this IDA Record Document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A137 335 / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Operational Software Technology Working Group Report (IDA/OSD R&M Study)	5. TYPE OF REPORT & PERIOD COVERED Final July 1982 - August 1983
	6. PERFORMING ORG. REPORT NUMBER IDA Record Document D-38
7. AUTHOR(s) LTC Lawrence E. Druffel, USAF Working Group Chairman	8. CONTRACT OR GRANT NUMBER(s) MDA 903 79 C 0018
9. PERFORMING ORGANIZATION NAME AND ADDRESS Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, VA 22311	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Task T-2-126
11. CONTROLLING OFFICE NAME AND ADDRESS Office of the Assistant Secretary of Defense (MRA&L), The Pentagon Washington, D.C. 20301	12. REPORT DATE August 1983
	13. NUMBER OF PAGES 51
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) DoD-IDA Management Office 1801 N. Beauregard Street Alexandria, VA 22311	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

None

18. SUPPLEMENTARY NOTES

N/A

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

reliability, maintainability, readiness, operational software, Software Technology for Adaptable and Reliable Systems (STARS), failure compensating software, fault avoidance, fault tolerance

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document records the activities and presents the findings of the Operational Software Technology Working Group part of the IDA/OSD Reliability and Maintainability Study conducted during the period from July 1982 through August 1983.

DD FORM 1 JAN 79 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

IDA RECORD DOCUMENT D-38

OPERATIONAL SOFTWARE TECHNOLOGY WORKING GROUP REPORT

(IDA/OSD R&M STUDY)

Lt. Col. Lawrence E. Druffel, USAF
Working Group Chairman

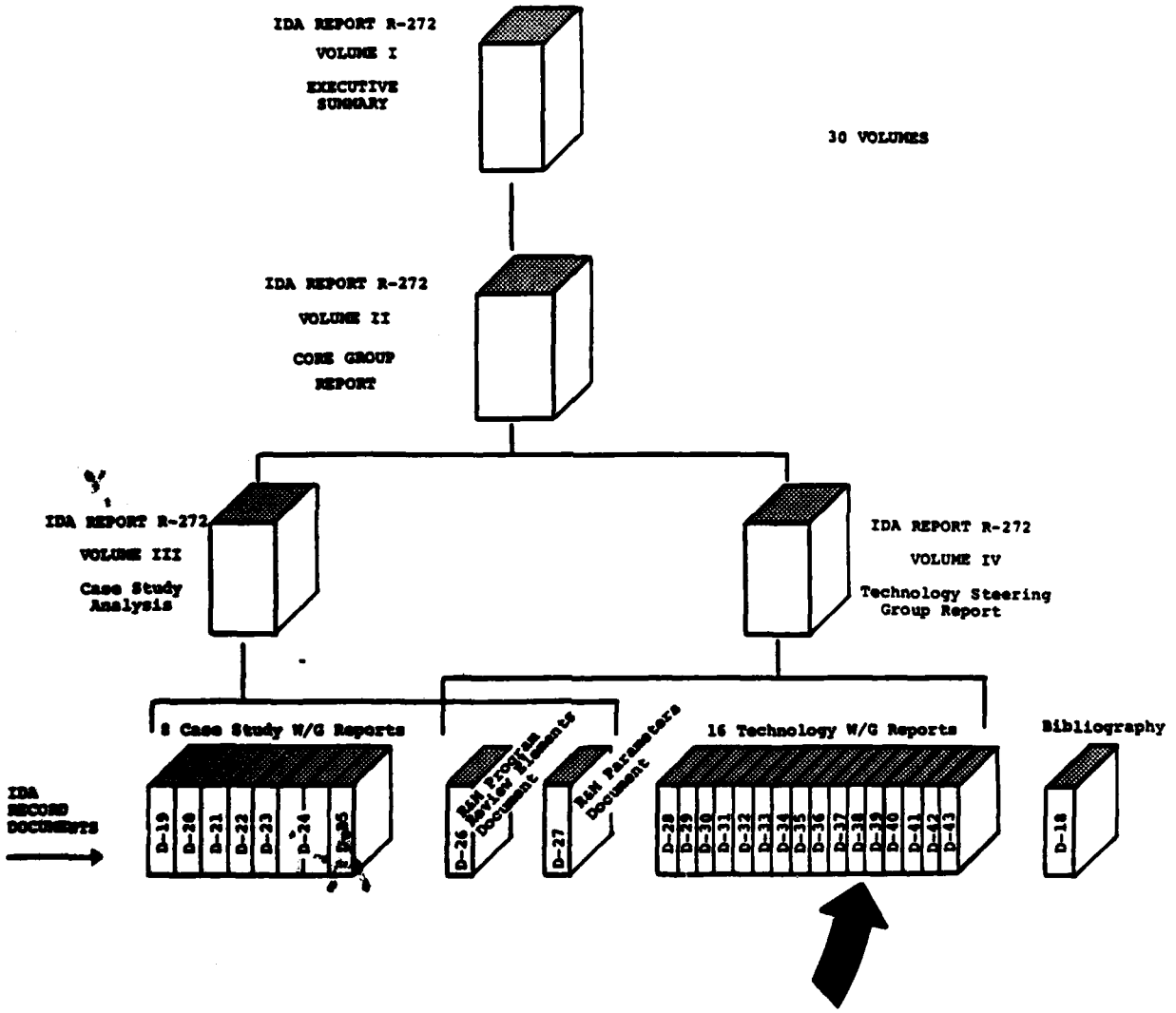
August 1983

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION
1801 N. Beauregard Street, Alexandria, Virginia 22311

Contract MDA 903 79 C 0018
Task T-2-126

RELIABILITY AND MAINTAINABILITY STUDY

— REPORT STRUCTURE —



THIS DOCUMENT (IDA Record Document D-38)

PREFACE

As a result of the 1981 Defense Science Board Summer Study on Operational Readiness, Task Order T-2-126 was generated to look at potential steps toward improving the Material Readiness Posture of DoD (Short Title: R&M Study). This task order was structured to address the improvement of R&M and readiness through innovative program structuring and applications of new and advancing technology. Volume I summarizes the total study activity. Volume II integrates analysis relative to Volume III, program structuring aspects, and Volume IV, new and advancing technology aspects.

The objective of this study as defined by the task order is:

> "Identify and provide support for high payoff actions which the DoD can take to improve the military system design, development and support process so as to provide quantum improvement in R&M and readiness through innovative uses of advancing technology and program structure."

The scope of this study as defined by the task order is:

> To (1) identify high-payoff areas where the DoD could improve current system design, development program structure and system support policies, with the objective of enhancing peacetime availability of major weapons systems and the potential to make a rapid transition to high wartime activity rates, to sustain such rates and to do so with the most economical use of scarce resources possible, (2) assess the impact of advancing technology on the recommended approaches and guidelines, and (3) evaluate the potential and recommend strategies that might result in quantum increases in R&M or readiness through innovative uses of advancing technology.

The approach taken for the study was focused on producing meaningful implementable recommendations substantiated by quantitative data with implementation plans and vehicles to be provided where practical. To accomplish this, emphasis was placed upon the elucidation and integration of the expert knowledge and experience of engineers, developers, managers, testers and users involved with the complete acquisition cycle of weapons systems programs as well as upon supporting analysis. A search was conducted through major industrial companies, a director was selected and the following general plan was adopted.

General Study Plan

- Vol. III • Select, analyze and review existing successful program
- Vol. IV • Analyze and review related new and advanced technology
- Vol. II (• Analyze and integrate review results
  (• Develop, coordinate and refine new concepts
- Vol. I • Present new concepts to DoD with implementation plan and recommendations for application.

The approach to implementing the plan was based on an executive council core group for organization, analysis, integration and continuity; making extensive use of working groups, heavy military and industry involvement and participation, and coordination and refinement through joint industry/service analysis and review. Overall study organization is shown in Fig. P-1.

The basic technology study approach was to build a foundation for analysis and to analyze areas of technology to surface: technology available today which might be applied more broadly; technology which requires demonstration to finalize and reduce risk; and technology which requires action today to provide reliable and maintainable systems in the future. Program structuring implications were also considered. Tools used to accomplish

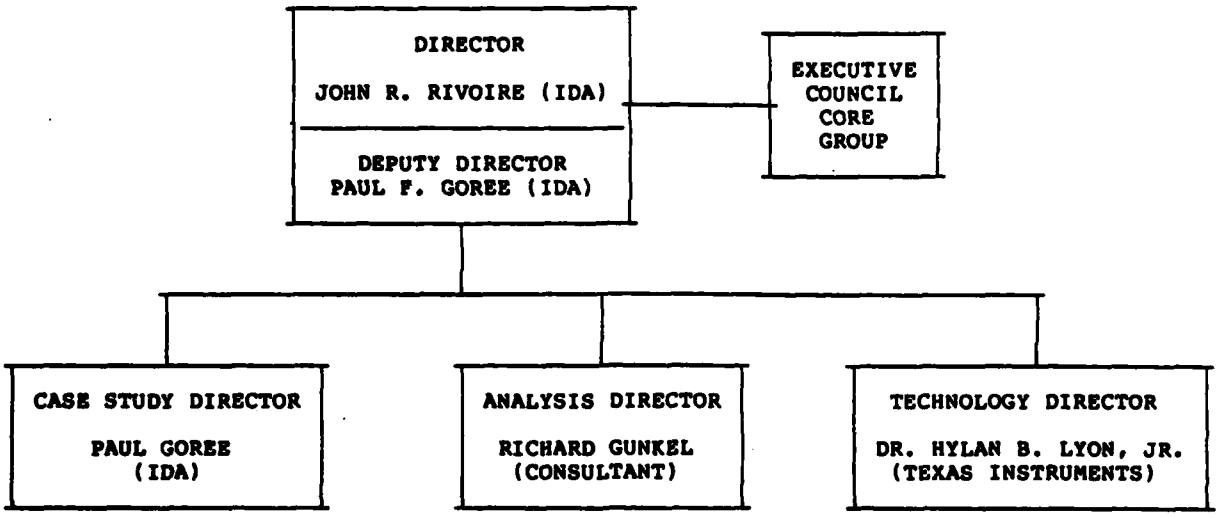


FIGURE P-1. Study Organization

this were existing documents, reports and study efforts such as the Militarily Critical Technologies List. To accomplish the technology studies, sixteen working groups were formed and the organization shown in Fig. P-2 was established.

This document records the activities and findings of the Technology Working Group for the specific technology as indicated in Fig. P-2. The views expressed within this document are those of the working group only. Publication of this document does not indicate endorsement by IDA, its staff, or its sponsoring agencies.

Without the detailed efforts, energies, patience and candidness of those intimately involved in the technologies studied, this technology study effort would not have been possible within the time and resources available.

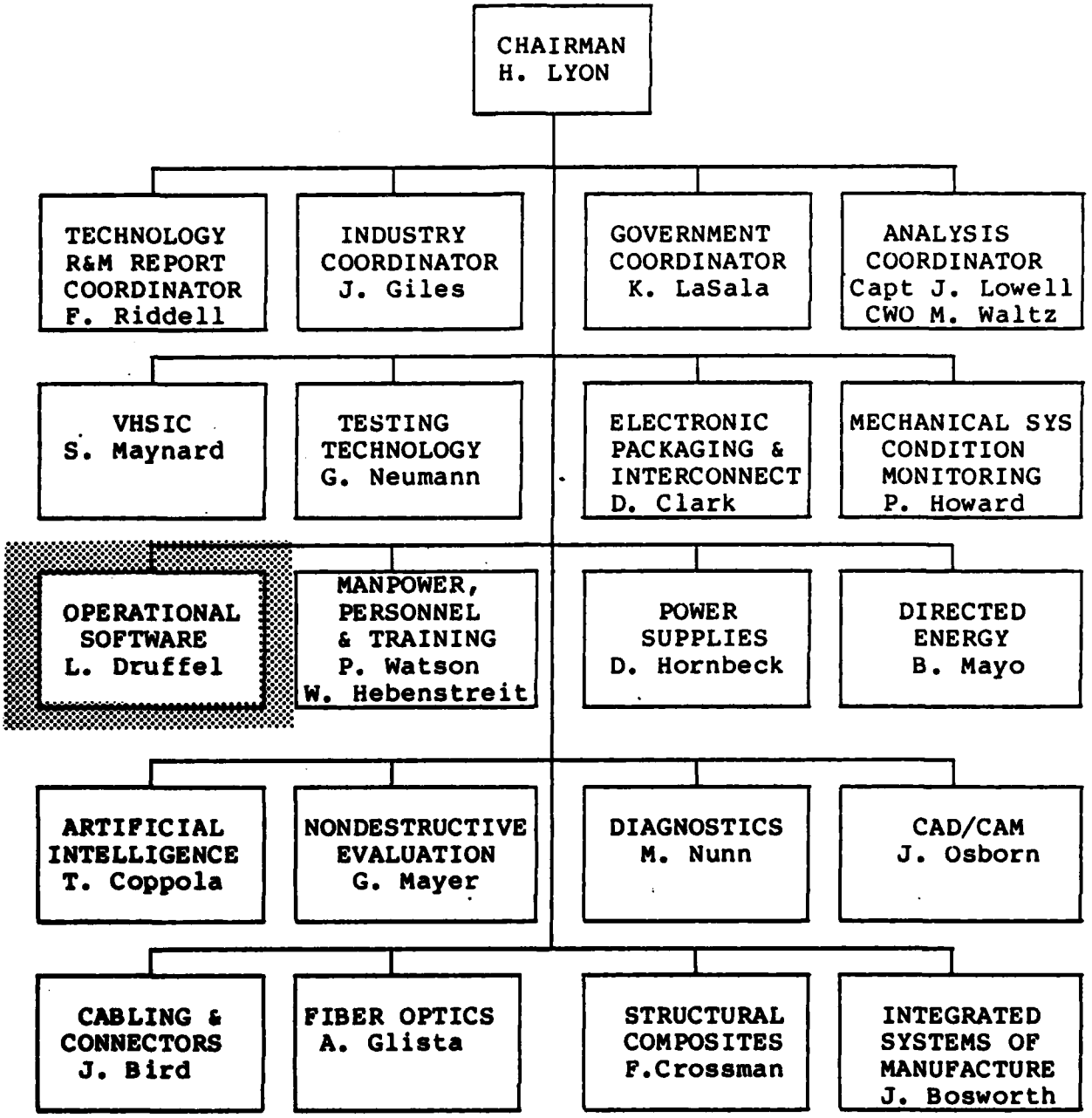


FIGURE P-2. Technology Study Organization

OPERATIONAL SOFTWARE TECHNOLOGY

STUDY REPORT

July 20, 1983

VOLUME IV, PART N OF THE OSD/IDA SYSTEMS

RELIABILITY AND MAINTENANCE

STUDY

A

Table of Contents

1. INTRODUCTION

This report addresses the potential impact of software technology on system reliability and maintainability (R&M) and on operational readiness. A considerable amount of attention has been given to software during the past year in preparation for the Software Technology for Adaptable and Reliable Systems (STARS) program. Consequently, much of the material in this report was adopted from the STARS program [3 , 4]. This is a natural choice because the objectives of STARS largely overlap the objectives of this study effort.

The Operational Software Technology study group was commissioned late in comparison to the other groups studying aspects of system R&M. Only one and one half months were originally scheduled for the study effort. This was insufficient time to adequately study and analyze such an extensive subject area. Consequently, what follows is not an unbiased study of the impact of software on system R&M. Rather it is the collected opinion of people who have studied a broader range of software issues.

The role of software in system R&M and readiness can be viewed from several perspectives. For example, software is a component of most modern weapon systems and affects system reliability. Software also is a component of many tools used for reliable construction of systems, e.g., CAD/CAM. This study focused on several important views of software. Sections 2 through 5 define the bounds of the study. Section 6 presents the results and findings, and section 7 the recommendations. The remainder of the introduction summarizes the importance of software in weapon systems. The introductory material was adapted from [3].

1.1 The Growth in Prominence of Software

Computers are an integral part of DoD weapon systems. Virtually every system in the current and planned inventory makes extensive use of computer technology. Computers are integral to our strategic and tactical planning. They control the targeting and flight of missiles; they coordinate and control the sophisticated systems within high performance aircraft; they are at the heart of the defense of carrier battle groups; and they integrate the complex activities of battlefield command. The military power of the United States, as we know it today, is inextricably tied to the digital computer.

Over the past twenty-five years, the weapon system computer has evolved from a role of minor importance to one of major importance. This trend has been accelerated in recent years by the microelectronic revolution which has been steadily improving the cost/performance ratio of digital computers. The amazing improvement in weight and power requirements of computers has made it possible to use computers in weapon systems in ways not envisioned five years ago. The improvement has been so great that embedded computer systems are now a primary means of introducing new capability and sophistication into our weapon systems.

Software for these embedded computers consists of computer programs and computer data, which are integral to weapon system operation, training, or support. Typically, such software has real-time constraints, performing both a component control function and an integration function for the system (such as inter-component communication and control).

In early uses of embedded computer systems, the functional capability of the weapon system was embodied largely in weapon system hardware (sensors, control devices, etc.) with the computer performing ancillary functions. As the digital system evolved to control not only the central function but also the inter-system communications, the role of software shifted from an incidental role to that of implementing the essential system functions. The

hardware is now simply the means by which those functions are executed. Today, it is necessary to understand the software in order to understand the capabilities and workings of the weapon system.

Software is a high leverage component of the system. Even though development costs for computer hardware have continually decreased, the cost of hardware still dominates weapon system acquisition due to the recurring costs of purchasing multiple components during the production phase. Thus, although software has increased in function and complexity, and in its importance to virtually every modern weapon system, it is not always managed as rigorously as the hardware portion of the system. This lack of rigor can result in cost and schedule overruns. More importantly, the software may have subpar reliability and maintainability characteristics which will affect adversely the operational effectiveness and life cycle costs of the entire weapon system.

Because software is a critical path item and usually the last component to be completed in weapon system development, less than fully effective management of the software development effort can cause late system delivery. While a late delivery itself affects readiness, total system readiness can be impacted in other ways. For instance, system function to be provided by software might be dropped or deferred in an attempt to field a weapon system on schedule.

It is necessary to understand the nature of embedded computer software to understand the magnitude of the challenge faced by DoD. The software controlling some DoD embedded computer applications is among the most complex human-designed systems in the world. Embedded computer software exhibits characteristics which differ markedly from other types of software. For instance, business applications software generally uses numerical and alphabetic input and generates files, printed reports, or displays. Embedded computer software usually uses analog (converted to digital) and/or digital input from a variety of sensors and sources, and generates digital control output to complex weapon system components or status information for human use.

In comparison with typical business applications of computers, DoD embedded computer software is often required to provide considerably more complex functions. The complexity of these functions is compounded by factors such as size, number and type of interactions, real-time response requirements, and distributed systems issues. In addition, this software is usually designed and developed in parallel with hardware (target computer and other subsystems). Further, to satisfy production schedules, hardware is often frozen earlier and changing requirements are deferred to the software. This results in a development activity of higher risk than typical business applications. The critical nature of embedded computer software means that reliable performance is of much higher priority than it is for business applications. Failures in embedded computer software involve time and dollars, but more significantly, they involve military weapon systems whose failure can result in the loss of human life.

The management approach, design techniques, and development process for DoD software have many similarities to those used for business systems, but the complexity demands a rigor and scope far exceeding that required of business systems. To characterize the size and complexity of embedded computer software from another point of view, it is useful to consider its cost. The software development costs for several weapon systems, including training devices and automatic test equipment, has exceeded $100 million, e.g., B-1B, E-3A, AEGIS, and Safeguard. Such development projects require hundreds of people applied over a period as long as five to ten years and support periods two to four times that long. The magnitude of these efforts ranks DoD embedded computer software among the most complex endeavors undertaken anywhere.

1.2 The Relation of Software to the U.S. Military Mission

By exploiting the flexibility of software in development of its modern weapon systems, DoD has elevated the importance of embedded computer software. Potentially, a function embodied in software may be modified, to

improve a capability or to meet new threats, more quickly and less expensively than the comparable function embodied in hardware. The Air Force F-111 program illustrated this point. The following table compares similar capabilities (additional offset aim pointer and updated weapon ballistics) implemented through hardware on the F-111 A/E and in software on the F-111 D/F. The savings in dollars and deployment lead time in the digital F-111 D/F are striking. Given an existing software support facility, the savings due to making the changes via software rather than hardware have ratios of about 50:1 in cost and 3:1 in time.

Modification	Via Hardware	Via Software	Cost/Time Ratios
#1	$5.28M/42 Mo.	$0.10M/16 Mo.	52.8:1/2.6:1
#2	$1.05M/36 Mo.	$0.02M/10 Mo.	52.5:1/3.6:1
#3	$8.00M/78 Mo.	$0.02M/15 Mo.	400:1/5.2:1

Another example of the advantages which can be derived from changing software without a physical change to the hardware was the reprogramming of the Minuteman III missile [1]. By modifying the software, engineers were able to improve the accuracy of the system without expensive hardware change. The change was designed and implemented in all Minuteman III missiles for approximatedly $4 million, a relatively low cost for the performance improvement.

In a more recent example, it has been reported [2] that the Sea Wolf missiles, which were just coming into service with the British fleet when the Falklands conflict began, had problems with their control and guidance systems caused by the missile exhaust plume interfering with the television guidance system. This problem was corrected by modifying the software while the ships

carrying the missiles were at sea and in combat. The software change enabled the missile to fly offset from the direct line of sight until near the target. The Sea Wolf missiles are credited with downing six enemy aircraft with additional aircraft crashing while taking evasive maneuvers.

The relative adaptability of software has made it an important factor in modern weapon systems, however that same adaptability has put software in a position to impact the military mission adversely if it is not managed properly. This impact manifests itself as cost escalation, lengthened deployment lead time, and operationally unreliable systems. There is evidence that software is having this effect and that the problem is becoming more severe with time.

Software is often the critical path item in weapon systems deployment because many of the requirements and design changes that occur during development are absorbed by the software. It is also the integrating element of the system. This again is a consequence of the adaptability of software. Therefore, the efficiency of the software development process and the mechanisms to react to change are directly related to the deployment lead time. If the years it takes to develop a software system can be reduced to months, and if the months it takes to implement major revisions can be reduced to weeks or even days or hours, the U.S. will significantly increase its ability to react to new threats or to pose unanswered threats to our adversaries.

The operational effectiveness of today's weapon systems are of critical concern. The degree to which the embedded systems help meet the operational need and perform reliably is directly related to the quality of the software. Recent experience where systems have not met the need, or have been faulty due to software, underline the importance of learning how to better develop complex software. This is especially true where software relates to the control and delivery of nuclear weapons.

2. STUDY GOAL AND OBJECTIVES

The goal of this study is to determine how a quantum improvement in system R&M and in system readiness can be achieved, or partially achieved, through advances in software technology. When viewing software as a system functional component, maintainability and operational readiness are dependent on the adaptability of the software. The speed with which we can react to new threats or repair software faults is directly tied to the adaptability of the software.

The following objectives support the goal of the study:

a. Identify and analyze the ways that software can impact system R&M and readiness.

b. Survey the state-of-the-art in software technology relevant to system R&M and readiness.

c. Identify the points in the acquisition process where software considerations become important in regard to system R&M.

d. Make recommendations for the exploitation by DoD of software technology as a means to improve system R&M and readiness.

3. SCOPE

There are several perspectives of software regarding its potential impact on system R&M and readiness. These perspectives can be summarized into three categories.

a. Software as an operational component of the system - this category includes mission critical software (software that performs an important system function). As one of several system components, this category of software directly affects the reliability of the system.

b. Failure-compensating software - this category includes operational software that anticipates and/or compensates for the failures of system components thereby preventing system failure.

c. Software supporting the development or maintenance of reliable components - this category of software includes software as part of tools that aid the reliable development or maintenance of systems.

Some of the software in category (c), such as CAD/CAM or ATE, are addressed directly or indirectly by other study groups and, therefore, are not addressed in this study. The scope of this study is limited to category (a) and (b) software and that part of category (c) supporting the development or maintenance of category (a) and (b) software. This type of category (c) software is referred to collectively as "support software."

4. ISSUES

One of the primary issues facing the study group is the meaning of software reliability. The definition of hardware system reliability is reasonably stated. Probabilistic techniques exist for estimating system reliability based on the reliability of its hardware components. A hardware component fails when it wears out and, therefore, the probability of failure can be estimated from history and/or design data. Software failures are not due to "wear and tear". Software failures occur because of undetected faults. A failure occurs when a logic path on which a fault resides, is executed with a certain set of data values. For large operational software systems there are many undetected faults. In some cases, the system operates for years before the proper set of circumstances occur to cause a failure due to a latent fault. In order for the software component "reliability" to be factored into a system reliability estimate, a meaningful definition of software reliability must be developed.

The definition of software maintainability also is complicated by the lack of a clear analogy to hardware component maintainability. Mean time to repair (MTTR) is a standard measure of hardware component maintainability. MTTR is the mean time to repair or replace a worn or broken part. Since software doesn't wear out or break, two possible interpretations might be given to "repair". One is to use the measure to indicate the average time required to restore the software to operation after a failure. This interpretation assumes no correction of the underlying problem that caused the failure. This situation may occur many times in an operational system before the error is properly characterized. Indeed, some systems are operated for years with failure-causing software faults. The other interpretation is to measure the average time required to find and fix software faults that cause failures. The fix can range from a single instruction change to a major redesign and reimplementation of a software component.

In order to evaluate quantitatively the impact of an operational software component on system R&M, definitions of software reliability and maintainability metrics must be provided which can be mathematically combined with corresponding metrics for hardware components to estimate overall system reliability and maintainability. Section 6 of this report points out that no practical definitions for software R&M exist today. Despite this, the report discusses various opportunities for qualitative improvements in software R&M.

5. APPROACH

The study group on operational software was initiated after the other study groups were well underway. Only one and one-half months were provided under the original schedule to perform the study. This was not sufficient time to initiate and conduct an adequate study of a subject area of this breadth. Therefore, the following strategy was adopted:

a. Since the Software Technology for Adaptable and Reliable Systems (STARS) program addresses reliability of software as one of its major objectives, the study and planning material developed under STARS was used as the primary source of information regarding software as an operational component of systems.

b. In the area of failure avoidance software, a quick survey of the literature and the commercial marketplace was conducted to identify the bounds of the state-of-the-art, and to identify current examples of the application of this technology in government and industry.

c. The study report material was assembled by a small group. Lt. Col. Larry Druffel was the study group leader. He is the former director of the Ada Joint Program Office and of the STARS program. Technical support was provided by Messrs. Andrew Ferrentino, Barry Perricone and John Sapp of Software Architecture and Engineering, Inc. (Software A&E).

d. Very experienced software professionals from the Services and industry were used to review and critique the material, drawing from their own experience in software reliability. Although most of the comments provided are reflected in the report, time did not allow for complete resolution of all issues raised by the reviewers. This report cannot be viewed as representing a consensus of the reviewers. The review team included:

Joseph Cavano
Software Engineering Section
Rome Air Development Center
Griffiss AFB, New York 13441

Richard DeMillo
School of Information and
Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Owen McOmber
Naval Embedded Computer Program Office
Headquarters, Navy Material Command
Washington, D.C. 20360

Samuel Redwine
Information Systems Department
MITRE Corporation
1820 Dolley Madison Boulevard
McLean, Virginia 22102

William Riddle
Software Design and Analysis
1670 Beak Mountain Drive
Boulder, Colorado 80303

Dennis Turner
Communications Software Support Division
CENTACS, DRSEL/TCS-SSD
Building 1210
Ft. Monmouth, New Jersey 07703

Robert Westbrook
Tactical Software Engineering Division
Naval Weapons Center
China Lake, California 93555

AIAA Software Systems Technical Committee:

Robert Jones	Hughes Aircraft Company Ground Systems Group P.O. Box 3310, Bldg. 618(B218) Fullerton, California 92634
Merlin Dorfman	Lockheed Missile and Space Company P.O. Box 504, MS62-22 Sunnyvale, California 94086
Herbert Hecht	SOHAR Incorporated 1040 South LaJolla Avenue Los Angeles, California 90035
Dennis Bunney	Systems Development Corporation 2500 Colorado Avenue Santa Monica, California 90406
Richard Van Tilburg	Hughes Aircraft Company Ground Systems Group P.O. Box 3310, Bldg. 618(B218) Fullerton, California 92634
Frank McGarry	NASA Goddad Spaceflight Center Code 582.1 Greenbelt, Maryland 20770

6. RESULTS AND FINDINGS

The study addressed three categories of software relating to system R&M and readiness: software as an operational component of a system, support software, and software that compensates for system component failures. Software as an operational system component is addressed first in Section 6.1. The difficulties faced in attempting to measure software R&M are discussed. At this time, an effective technique for measuring software reliability or maintainability does not exist. This leads to a discussion of fault avoidance in section 6.2. Support software is a key aspect of this subject. Although a great improvement can be made in reducing the fault content of software, complete elimination of faults is not yet feasible. Fault tolerant techniques can compensate for latent faults in software. These techniques are summarized in section 6.3. Finally, fault avoidance and fault compensation considerations in regard to the system acquisition cycle are discussed in section 6.4.

6.1 Measuring Software Reliability and Maintainability

To quantify the impact of software component reliability on system reliability, a metric for software reliability must be defined which can be combined with a corresponding metric for hardware components. A common measure of hardware reliability is mean time between failures (MTBF). The causes of hardware failures can be grouped into three categories. The first category includes failures due to environment anomalies, e.g., a loss of the power supply. The second category includes faults introduced in the design or production process. The third category includes failures resulting from operational stress over time, i.e. a part wears out or breaks. For operationally mature hardware, the third category is most prominent in computing MTBF. In this case, the MTBF is determined by the physical attributes of the hardware and can be estimated reasonably well.

Software is quite different from hardware. Software does not wear out, therefore it has no failures corresponding to the third hardware category. The closest analogy would be a software failure due to an overload condition, e.g. real-time processing of sensor inputs failing to keep pace with the rate of sensor data input; however the failure involves no "worn or broken" parts. Most software failures are caused by faults in the software introduced during software development or modification. There are models of software reliability available today [10, 11], but none that are sufficiently mature to be effective in predicting software reliability. Given the nature of software failures, as discussed in section 6.1.1, it is possible that MTBF is not a suitable measure of software reliability.

There are two ways of viewing software maintainability. One view is that maintainability is a measure of the time required to restore a software system to operational status after a failure. Time to restore is dependent on many factors such as the degree of damage to files, if any, and the degree to which recovery aids are built into the software. Many of these issues are addressed under fault tolerance in section 6.3.1. The second view is that maintainability is a measure of the time required to find and fix a fault in the software that caused an operational failure. Although maintainability viewed as time to restore the software to operational status is a more useful measure for the purpose of computation of system availability, the view of maintainability as time to repair was the primary focus of the study effort, because this view results in the upgrade of the system to full operational capability.

The issue of software maintainability is more tractable than software reliability in the sense that well-engineered software may attain an operational equilibrium where fault fixes have a predictable lead time on the average. But as with reliability, software maintainability differs significantly from hardware maintainability. With a hardware failure, the failing part is patched or replaced. It may eventually fail again. When software fails and is fixed, it is unlikely it will fail in the same way a

second time (assuming good engineering practices). Also, it is often possible to work around software faults once the conditions that lead to execution of the faulty code are known, in which case the fault may be left in the software.

The basic hardware measure of maintainability is mean time to repair (MTTR). For repair of software, this measure is driven by the adaptability of the software. If the software is designed for change, the MTTR may be smaller than if it is not or if the design has degraded in integrity through many years of modification. The factors that affect adaptability are discussed in section 6.1.2

6.1.1 Software Reliability

In discussing software reliability it is useful to make a distinction between faults, errors, and failures. A failure is the occurrence of a deviation of the external behavior of the software from that defined in the software specification. An error, or more specifically an erroneous state, is an internal software state which could lead to a failure under continuing operation. A fault is the cause of an error. Faults can result from environmental conditions, e.g. a bit "dropped" by the hardware. A software fault is incorrectly implemented code that, when executed with certain data values, causes an erroneous state. For example, an incorrectly coded algorithm is a fault. The execution of the faulty code that computes the algorithm and stores the result creates an erroneous state (i.e. the stored result of the computation). A failure may occur when the erroneous computational result is used to generate an external system event, such as the firing of a weapon.

The reliability of an operational software component is characterized by the number of uncorrected faults in the operational code, the probability that a fault will be encountered under the expected use patterns of the system, and the severity of the error resulting from the execution of the faulty code.

Quantitative prediction of any one of these factors in a software component is very difficult. Of the three factors, the probability of encountering a fault during execution and its severity is the more important issue in regard to reliability. The number of uncorrected faults can be misleading. One severe fault in a location where it is likely to be encountered may have more of an impact on reliability than one hundred faults of lesser severity in seldom used logic paths. Likewise, a fault which produces catastrophic results such as the failure of an automatic landing system is of more concern than one which produces a bothersome but benign situation such as an inappropriate warning indicator.

The probability that a software failure will be encountered in the operation of a system is dependent on the distribution of the faults in the code and the use patterns of the system. If the software was tested according to the expected use patterns, the system failures due to software faults will likely occur only in a seldom encountered use pattern. In this event, the decision might be made to leave the fault in the system rather than fix it if the system users can work around the use pattern without significant operational impact. The troublesome aspect of software faults in weapon systems is that many faults may go undetected until a critical mission which imposes different use patterns than peacetime exercises of a weapon system. Thus the rate of system failure due to software faults may increase at just the time that high reliability is essential.

As pointed out earlier, not all faults encountered in the execution of a software component result in system failure. A classification scheme for software faults relative to failure would be very useful. Such a scheme might include origin, means of discovery, location, means of correction, criticality, type of computing or action involved, level of actual damage, and frequency. Steps can be taken in the design of software to lessen the impact of errors resulting from software faults or hardware faults. One such technique is called exception handling (this is a subpart of fault tolerance

to be addressed later). Software components in mission critical systems should contain these mechanisms. Ada provides explicit language features for exception handling which, if used properly, should lead to more reliable software components.

Determining the number of faults remaining in a software component at any point in time is a vexing problem. Several techniques are promising for the statistical estimation of the number of remaining faults but many questions remain to be answered. Fault seeding [13] is one such approach but its success depends on the type of faults that are seeded and where they are placed in the software. A testing strategy which supports the fault seeding strategy also is necessary. Despite our inability to accurately predict the number of faults in a software component, there is some promising work evolving in measuring the relative likelihood that faults are present in a software component. One technique makes use of a software complexity measure to make such a judgement [14].

In summary, the reliability of software is dependent on the number and distribution of faults in the code, the operational use patterns of the system, and the severity of an error caused by execution of faulty code. Until practical methods of estimating latent faults in software can be found, until a fault classification scheme can be developed, and until error occurrence can be predicted, the definition and estimation of software reliability will not be practical.

6.1.2 Software Maintainability

Software maintainability and the contribution of software to system readiness are directly related to software adaptability. Adaptability is a measure of the effort required to change the software. The ease with which software can be changed is dependent on many factors. Four of the more important factors are discussed below.

a. Complexity - There is no widely accepted meaning of the concept of software complexity [14, 15], but it is generally agreed that as the number of components and the interrelations among those components grow, the software becomes more difficult to understand. For large systems (systems with many components), a way to reduce complexity (defined as difficulty in understanding) is by reducing interrelations. This may be accomplished through decomposition of the system into separate components or modules. Many systems today are designed this way but are still complex because the modularity technique used allows more interrelations than are necessary. These excessive interrelations make change an error-prone process because the effect of the change extends, through interrelations, beyond its intended scope.

b. Levels of Implementation - Many software systems are implemented at but one level, e.g. assembly language. There are higher level implementations possible. These include high order languages(HOL), table-driven software, and non-procedural languages, proceeding from lower to higher levels. If a change can be made in a system by changing a data table, potentially this change will be made faster and more reliably than a corresponding change made in assembly language code. If special tools exist to assist the table modifications, the change will be even faster and more reliable. Although many systems are implemented in an HOL today, few systems make extensive use of table-driven software and even fewer have application-specific non-procedural languages built into the development environment.

c. Traceability - The ability to change software easily is directly related to the ability to trace a new requirement to the parts of the design and the code it impacts (and vice-versa). This traceability depends on the degree to which the software is designed with traceability in mind and the quality of the documentation. Systems that have poor traceability characteristics are difficult to change.

d. Formal Design Recording - The design of most operational software is either not formally documented or is poorly documented. This results in software modifications that are evaluated, planned, and implemented at the detailed source code level. For complex real-time systems, this results in a change process that is longer and more costly than it should be. It also results in degradation of the design integrity and a corresponding increase in software complexity. To make software adaptation an efficient process, proper formal design documentation is needed. This documentation should clearly reflect the modularity of the system. It should present various levels of description detail. It should minimize the amount of documentation to be searched and understood to modify the software correctly. The design documentation should separate the underlying design concept from the transformations applied to optimize run-time efficiency.

As an illustration of the impact of these characteristics consider the following real-life situation that is not unusual for many software systems today. A real-time software system of 150,000 source lines of code is written entirely in assembly language. The design exhibits little modularity and interrelations are numerous. There is little design documentation and traceability is very difficult. In short, it exhibits none of the characteristics listed above for adaptability. The consequence is that it requires 40 people to maintain that software and no major changes are made to it because of the difficulty of making even simple changes reliably.

Few systems are developed with adaptability as a development requirement. However, even for those software systems that are designed such that change is possible with relative ease, the adaptability of these software systems degrades with the number of changes applied. This phenomenon described by Lehman and Belady [12] is akin to entropy in that the complexity of a system tends to increase exponentially with time due to the continued modifications required for operational software systems. Therefore, not only does a software system have to be developed with adaptability as a goal, but modifications made over its life cycle also must be aimed at retaining adaptability characteristics in order to forestall the effects of "entropy".

Maintainability is defined for the purpose of this report as the average time required to find and fix software faults. The time can vary significantly for a given software component depending on the characteristics of the fault(s) causing the failure. Some faults are easy to find and may require the correction of a single line of code. Other faults may be very difficult to isolate and require major redesign and code development to fix.

The maintainability of a software component is dependent on many factors. One factor is the adaptability of the software as discussed above. Another is the tooling available to support find-and-fix activities. Today, software maintenance is labor intensive in a marketplace where professional shortages are severe. To reduce the impact of labor shortage and to decrease the MTTR, tools must be introduced to aid debugging, data reduction and analysis, and testing among other activities. This tooling should include capabilities built into the operational software component for real-time operational analysis and diagnosis (possibly at a remote site).

Testing is another factor in software maintainability. The testing required to ensure that a software fix corrects the fault of concern without introducing new faults is called regression testing. A regression test strategy determines what subset of the original test set for the system must be repeated because of a fix. A good strategy leads to the minimum subset that demonstrates correctness thus minimizing the impact on time to repair. The size of the test set does not correlate necessarily with the number of modules modified. A small change to one module can have a ripple effect through the software component, requiring testing of many modules, including the one modified. Current design techniques can limit this ripple effect but they are not yet widely used by software production groups.

In summary, software maintainability is a measure of the time required to find and fix faults. This time is dependent on the design of the software, its adaptability, the tooling available to support the process, the regression testing required and many other factors. For most software today, these

factors are not managed well, leading to a great variability in the time to find and fix a fault. Therefore an MTTR computed for software may be misleading, especially if used in a system context to compute system availability.

6.2 Software Fault Avoidance

Given that latent software faults are the primary cause of operational software component failures, the reliability of operational software can be improved by reducing the run-time existence of such faults. Activities with this aim are referred to here as fault avoidance approaches. One such approach is fault identification through testing. Testing has proven to be a valuable technique for fault identification. Many improvements can be made in how software is tested today, however testing has limitations in regard to fault avoidance. Testing is discussed in Section 6.2.1.

Another approach to fault avoidance is to make no errors in the design and implementation of an operational software component during its development. Under a strict interpretation, this is not feasible for large systems, but if we interpret this to mean the identification and removal of development errors soon after they occur (i.e. within days or weeks), then it becomes a very practical objective. To realize the objective requires a high quality development process with supporting techniques and tools. These considerations are discussed in section 6.2.2.

6.2.1 Software Testing

Testing is the dynamic execution of a software component with known inputs in a known environment to obtain the expected response. If an unexpected response is obtained, the software is presumed to contain a fault. Therefore, testing provides a means to identify the presence of faults.

Historically, testing has been the primary means of fault identification during software development. Although testing is a powerful aid to fault reduction, it cannot be expected to produce fault-free software because the combinations of possible inputs, logic paths, and asynchronous events are astronomically large even for small embedded systems. Understanding this limitation, with a well-conceived testing approach, software faults can be reduced to an operationally acceptable level. Some of the important attributes of a well-conceived testing approach are discussed below.

a. Specification - to test software properly, a basis of determining expected software behavior must be available. The expected behavior should be explicitly defined in a specification (e.g. requirements specification or functional specification). The specification must be complete and unambiguous to the degree that the important operational behavior of the software component is explicitly defined.

b. Planning - developing test plans and procedures for a complex software component can be a sizable effort. To develop adequate test plans and procedures, early involvement in the development process is necessary. Test planning must begin early, as called for by DoD Directive 5000.3, and continue in parallel with software development.

c. Coverage - Since it is infeasible to exhaustively test a software component, the objective of test planning is to define a minimal test set, or coverage, that will provide an operationally acceptable level of remaining software faults. There are several test strategies which can support this objective. Functional, or black box, testing is the development of tests using the functional specification as a guide. Structural, or white box, testing uses knowledge of the software design or implemented code to define the test coverage. For a complex software component a combination of functional and structural techniques is desirable.

d. Levels and objectives - testing should be carried out at various levels of development, i.e. unit, integration, and system level testing. At each level, the objectives of the testing will differ. For example unit testing may have as an objective to exercise every instruction whereas integration testing may focus exclusively on unit interfaces. The objectives of the various levels of testing in combination should provide an adequate test coverage strategy.

e. Administration - administration of the test activity includes test scheduling, results reporting, trouble report tracking, software change control, and regression test planning among others. Effective management of the testing activity is essential to testing. The most difficult aspect of administration is the determination of how much testing is enough. In many cases today this decision is made on the basis of schedule and budget. Techniques for estimating the reliability of the system based on test results would allow for more rigorous judgement of how much is enough. Such techniques are available today [17] but they lack the rigorous assessment of software reliabilty required for confident decision making.

f. Support Software - Effective testing of embedded computer software components requires a range of support software (or tools). These include unit test drivers, coverage analyzers, environment simulators, and tools to support the administrative activities.

There are few software systems today that have been exposed to test activities with the above characteristics. A great improvement can be made in the reliability of software with the imposition of a rigorous test process. DoD has recognized this and has initiated the Software Test and Evaluation Project (STEP) [16]. The goal of STEP is to develop enhanced policy guidance for DoD components in reference to test and evaluation of embedded computer software. STEP also intends to stimulate development of techniques and tools to support software test and evaluation.

6.2.2 The Software Development Process

A comprehensive approach to fault avoidance must go beyond software testing. The software development process must be oriented to the removal of faults shortly after they are introduced - during requirements definition, design, coding, and integration. Two useful techniques in this regard are formal reviews and static analysis. Formal reviews of technical products by technical peers in a well structured format has proven to be an effective means of early fault removal. Static analyses are methods for identifying faults by examination of the detailed design (expressed in a formal language) or by examination of the source code. These techniques can be automated.

A development process that supports early removal of faults is characterized by an integrated set of techniques and tools supporting all aspects of the process. It generates requirements, design and code products that are well structured for review and analysis. Most software today is not developed in this way. There are many well-documented difficulties with current software development approaches [3]. The DoD STARS program is geared to addressing these problems.

6.2.2.1 Software Engineering Techniques and Support Software

There is a significant gap between the state-of-the-art and the state-of-the-practice in software development as evidenced by the software engineering techniques and support software used in the DoD community (government and industry) today. Modern techniques such as formal requirements analysis, stepwise refinement, information hiding, data abstraction, structured programming, and static analysis are not used widely. The support software that is used centers on code generation with little or no automated support of requirements, design, or management activities. The development process is labor-intensive.

Although the state-of-the-practice can be greatly improved with available technology, there are two factors that resist this improvement. One is investment. It takes a great deal of investment to introduce new techniques and tools. The existing software must undergo costly re-engineering in order for the new technology to be effective, and new computers and support software may have to be procured. The second factor is people. In order for the new technology to be used effectively, the skills of the software professionals have to be improved, and new methods must be learned.

DoD is taking an important first step by introducing a standard, modern programming language, Ada, and an integrated support system called the Ada Programming Support Environment (APSE). The APSE will be introduced in an evolutionary manner. The initial capability, called the Minimal APSE (MAPSE), will support code generation. Tools to support requirements analysis, design, verification, testing, and management will be added over time. Through standardization of the interfaces, it will be easier to promulgate new techniques and tools into software development environments. Both the productivity of development groups and the reliability of the resultant software will be improved. Ada provides features supportive of data abstraction, information hiding, exception handling, and reuseable components. All of these are important in developing more reliable and adaptable software.

The APSE is the first major step in improving software engineering environments in DoD. It will provide the means to manage effectively the total information base associated with software development. Software, descriptive data, and management data all will be maintained in the APSE integrated data base. The APSE tools will use the integrated data base and will be accessed through a common command language. The introduction of integrated support environments that provide automated aid to all activities of the development process will positively impact software reliability and adaptability.

6.2.2.2 Standard Components

Another means of increasing the reliability and adaptability of operational software components is the use of tailorable, reuseable standard components. These might include real-time operating systems, data base management systems, message handlers, display formatters, and report generators, among others. Today we tend to custom build these components with each new weapon system even though the function is largely the same. Thus, each new incarnation goes through the same shakedown process as reliability stabilizes. If tailorable, standard components were used instead, the reliability of these components would be much better at the outset. At the same time, productivity and adaptability would be improved.

6.3 Software as a Means to Compensate for Component Failure

Software can be used to improve the reliability of a system by compensating for failures in system components. One approach, known as fault tolerance, is to use software to compensate for failures during system operation. Another approach is to redesign the system such that a failure-prone hardware component is replaced by software or by a combination of software and hardware. These two approaches are described in the following sections.

6.3.1 Fault Tolerant Systems

Fault tolerance is an attribute of a system that allows it to continue with its intended operational behavior after the occurrence of a fault. Fault tolerance has three key aspects: detection, damage assessment and containment, and recovery. These can be realized through a combination of hardware and software.

Fault detection is the automatic identification of a system component failure or erroneous state. Fault detection at various levels can be implemented in hardware or software. Hardware techniques include error detecting codes, disagreement detectors with majority voters, and built in test equipment [5, 6, 7]. Many of the hardware techniques such as disgreement detection, can be implemented in software. Software also can be used to monitor for certain conditions in an on-line mode.

Having detected that the system is in error, it is necessary to determine how much of the state of the system has been corrupted before recovery procedures are invoked. The assessment of damage is made based on knowledge of system structure and the flow of information in the system, and/or the outcome of additional error checking.

Once the extent or damage is estimated, steps may be taken to ensure that no further damage is inflicted on the system state. This may require the switching off-line of hardware components, the suspension of a software component, or the cessation of operations against a data base.

Recovery activities to compensate for the fault are initiated after damage containment is completed. Recovery techniques can be described in terms of three classes:

a. Full recovery - the system is returned to a state that existed prior to the fault. This usually requires redundant, reconfigurable hardware, software, and/or logical file components, periodic saving of the system state, and transaction logs to back out bad transactions from a file or to recreate the state of the system at failure from a saved state.

b. Fail soft - sometimes called graceful degradation, the system is returned to a fault-free state but without the service of all its components. Many of the same attributes of full recovery systems, such as redundant hardware and software components, are needed.

c. Fail Safe - the remaining functioning components are not sufficient to carry out minimal system activities and the system is safely shut down without damaging stored information or the non-failed systems components. Also, interactions with other systems are properly terminated.

The fault recovery activities can be implemented in hardware or software. The advantage of software-implemented recovery is that fault tolerant systems can be built with off-the-shelf hardware components. The disadvantage is that the recovery software itself may be impaired by the fault.

When fault tolerant techniques are applied to a software component of the system, the implementation is more complex than that of a hardware component [18]. The primary reason for this is that redundant software components cannot be implemented through duplication. If software fails due to a software fault, it is likely that a copy of the software will fail also. Therefore, a redundant software component must be designed and implemented as a functionally equivalent but different version of the primary operational component. There are two techniques for the use of redundant software components - recovery blocks and inversion programming. Recovery blocks test output assertions and invoke the operation of a redundant component when an assertion evaluates as false. Inversion programming makes use of a majority vote of the output of concurrently operating redundant software components.

Fault tolerant systems currently exist in government and industry [8,9]. The spread of fault tolerant systems has been modest because they can be costly. The primary cost is due to redundant components required in many fault tolerant architectures. With the rapid decline in the cost of digital hardware, the cost effectiveness of fault tolerance as a means of increasing system reliability is on the rise.

Some examples of fault tolerant systems include the FAA air traffic control system, the Bell electronic switching systems (ESS), Apollo control systems, and airline reservation systems. DoD is also a major user of fault tolerant systems such as those employed in submarine fire control systems.

Most fault tolerant systems use a combination of hardware and software for detection, and software controlled recovery. The fundamentals of fault tolerance are understood well enough that fault tolerance features are built into many commercial computer hardware and systems software. So-called non-stop systems are commercially available with many computer manufacturers entering this marketplace. This emergence in the commercial marketplace is due both to the solidification of an acceptable fault tolerant paradigm and to the decreasing cost of hardware.

DoD can exploit fault tolerance as a means of increasing system reliability. To do this DoD must capture the well-proven fault tolerant design paradigms much as the commercial computer vendors are doing and factor these designs into weapon systems. The decreasing cost trends in digital components will make this feasible on an ever broader scale. DoD also should consider the implications of fault tolerance in its computer standardization efforts. For the future, research into better digital hardware to encompass some of the necessary detection and recovery logic will enhance DoD's ability to field fault tolerant systems.

6.3.2 Software-intensive Systems

Another application of software as a means of compensating failures in system components is to redesign the system, replacing the failure-prone component with software or a combination of software and a more reliable hardware alternative. This is illustrated best through an example.

Consider a system that relies on sensors for inputs and uses a computer to process the sensor data. If one of the sensors is unreliable, it can have a severe impact on the reliability of the system. A more reliable alternate sensor might be available but exhibiting more noise content in its readings than the unreliable sensor. An alternative design might be to use the noisier sensor but with an increased sampling rate coupled with software filtering

techniques to obtain readings of comparable accuracy to the unreliable sensor. This may require more processing power in the computer. Thus, in this example, the combination of a noisier (but more reliable) sensor, software filtering techniques, and more processing power might result in a more reliable system.

The opportunity for system reliability improvement through design alternatives that are more software-intensive is open-ended. Taking advantage of the opportunities requires system designers who understand the potential applications of the digital computer in weapon systems and who understand the opportunity for systems reliability improvement through software.

6.3.3 Human Engineering

An often overlooked component of weapon systems or other mission critical systems is the human "component". Many systems today are human-interactive. Therefore, it is possible for these systems to fail due to human error. Software plays a role in this type of system failure because it controls or is a factor in the human interface of many systems.

A related problem is that of data base integrity. System failures can occur due to incorrect data in a system data base. One source of erroneous data is the system user. Unless systematic techniques for input validation are implemented, system failure due to data base errors can be expected.

Human engineering relates to the effective design of system interfaces with the human component. For software, this is an area which has had insufficient focus. Much improvement can be made in the design of software controlled human interfaces regarding the manner in which information is presented to the human, the way information is entered by the human, and the way the software reacts to human errors. Emerging techniques from the field of artificial intelligence might be applied to make the interfaces more natural to the human and more resilient in the presence of human error.

6.4 Acquisition Considerations

Within the current state-of-the-art, the opportunities for system reliability improvement through software are many. These opportunities can be realized today if the proper actions are taken at the right places in the acquisition cycle. The majority of the opportunities for reliability improvement through software occur before DSARC III, namely in the demonstration and validation phase and in the full-scale development phase. The use of software intensive design and fault tolerance must be addressed in the demonstration and validation phase. The reliability of software as a component becomes critical in the latter part of this phase and during the full-scale development phase.

During the demonstration and validation phase, the primary focus should be on design issues as they relate to R&M. High risk reliability components should be identified. Alternative designs using software in place of the high risk reliability components then should be evaluated. If the projected system reliability is not adequate, fault tolerant design alternatives can be investigated.

Once a system design with adequate reliability is completed, the digital subsystems portion of the design should be analyzed to ensure that no design decisions have been made which will make development of reliable software components difficult. An example of this would be the choice of a digital computer with insufficient memory causing overcomplication of the software component. Mechanisms are needed in the acquisition process to ensure that these design issues are properly addressed in the demonstration and validation phase.

At the initiation of the full-scale development phase, there are several items which should be included in the RFP for, and contract of, the development agent. The RFP should be based on systems and software

specifications that are well defined to the degree that the significant design decisions of the previous phase are unambiguously defined. The contract should include specific guidelines on software engineering techniques and tools to be used in development. The procuring agent should have oversight that the contractor is properly employing these techniques and tools. In addition, the RFP and contract should explicitly define requirements for software reliability and adaptability, e.g., parameterization of constants. Predictable mission changes and possible system changes to adopt emerging technological innovations should be identified. Finally, the contract should state requirements on the degree of software quality assurance and verification to be performed. Acceptance criteria should be clearly and fully defined.

Once the full-scale development contract(s) has been awarded, the procuring agent must take an active role in review and approval of the software product as it progresses through development. This can be accomplished through PDR, CDR and QA audits. The acceptance criteria for the system must be applied rigorously with proper emphasis on the adaptability requirements.

7. RECOMMENDATIONS

The Operational Software Technology study has identified several ways that software can impact system R&M and readiness. Improvements in areas such as testing, reliability prediction, software engineering techniques and tools, and fault tolerance techniques can have a significant positive effect on system R&M and readiness. The STARS program plan addresses these areas and many more in a broad software context that includes R&M. Therefore, the primary recommendation of this study effort is for DoD to use a fully funded STARS program as the means to exploit software technology for the improvement of system R&M and readiness.

The STARS Program has been established to improve mission-critical defense software and to preserve the U.S. lead in this field. It will accomplish this by addressing a wide range of software-related problems that have become pervasive and serious throughout the defense community. Through a strong, focused initiative that involves the defense components, industry, and the software research community, the STARS Program will create the technology and establish the practices that will facilitate quicker development and support of software and improvement in its quality, adaptability, and reliability. Areas to be attacked initially include acquisition and project management, measurement, human resources, human engineering, systems technology, software support systems, advanced applications, and technology transfer.

STARS will be managed by a jointly-staffed program office under the USDR&E and supported by an executive committee reporting to the DUSD (R&AT), service and defense agency program management offices, a joint review committee, and a Software Engineering Institute. The Program, which was initiated in the Spring of 1983, is funded at $222 million over the FY84-88 FYDP.

Each of the eight STARS technical areas addresses aspects of R&M as discussed in this report. A short description of each technical area and its R&M focus is given below.

a. Support Systems - Support Systems focuses on the preparation and support of demonstrably effective software development and in-service support in DoD software-intensive systems. The term "environment" is used in its technical sense to connote a "core" set of basic tools and an integrated, extended set of support tools. The "core" (or the core environment) is relatively invariant over time (i.e., it evolves slowly), whereas evolution of the tools and toolsets is expected. The term "support system" encompasses both environments and methods. Activities which should be supported include the development of methods and tools to support testing, to support the design of adaptable software, to support the design of reuseable components, and to support the maintenance activity. The techniques and tools recommendations of STEP will be factored into the plan and should be supported.

b. Systems - Systems is concerned with the target system environment but includes some concern with its relationship to the support system environment. A target system environment is the configuration of systems software and hardware in which the applications software operates. Improvements in the overall quality of defense systems depends upon a corresponding increase in the quality of the underlying systems software and hardware. This in turn requires that methods, tools, and knowledge to make effective use of the advanced systems technology be developed and placed in the support systems environment. Some of the R&M concerns which should be supported in this area are fault tolerant computer architectures, standardizing system component interfaces, definition of system development methods that foster high reliability, and research into reliability prediction. Systems issues relating to maintenance should be addressed also including considerations of remote diagnostics, embedded support tools, the logistics of fielding modifications and supporting documentation, and communications connectivity of distributed support groups.

c. Acquisition - The Specific goals and objectives of the Acquisition Task Area are the improvement of all business and contract related policies and practices, attainment of a higher degree of uniformity in the application of acquisition policies and practices; and an improvement of the tools associated with the acquisition of systems and software in order to streamline, simplify and accelerate the acquisition process; and to foster a more effective DoD contractor relationship. Major emphasis is to be placed on computer software associated with "mission critical" applications and embedded computer systems, and the integration of this software with the surrounding hardware. A thorough analysis of the acquisition process will be initiated and should identify areas were software R&M concerns should be addressed as suggested in section 6.4.

d. Measurement - Measurement is concerned with activities to develop models and metrics, creating and maintaining software data collection and analysis activities, supporting the use of the metrics and models during the total life cycle, and providing customized measurement support for the STARS program. Measurements which quantify software reliability and maintainability should be researched. Data collection activities should include reliability and maintainability.

e. Human Resources - The primary objective in the Human Resources task area is to "increase the level of expertise" and "expand the base of expertise" available to DoD. This is a direct attack by DoD and industry on improving human resources within the software field. Broadly stated, the objectives would be satisfied by defining software-related job knowledge, skills, and abilities; software-related education and training. The target audience includes personnel in software engineering and management. Activities will address the personnel shortages that are affecting the software maintenance groups in DoD. They also should address the need to teach software professionals new techniques in design relating to fault avoidance, fault tolerance and software adaptability.

f. Human Engineering - A primary objective in the Human Engineering Task Area is to incorporate human engineering principles into the design of all computer-based systems that interface with the human user. The characteristics that define good human engineering - such as ease of learning, flexibility and efficiency - cannot be added on at the end of system development but must be an integral part of the design from the beginning. There are a number of activities which must be performed during each development phase (i.e. requirements analysis, system specification, etc.) to insure this. What is needed is a methodology that focuses on human factors issues at all stages of the system development process. The design of human-friendly interfaces which exhibit human-error-tolerance will be one of the objectives.

g. Project Management - The overall objective in this task area should be to improve the practice of project management to contribute to the goals of: shorter schedules, higher quality products, greater cost effectiveness, better forecasting, and increased product knowledge transfer. The objective would be accomplished by producing and making available to project managers tools, methodologies, models, and training programs designed to achieve the goals. Activities will address techniques that foster management visibility into and control of the technical decisions that impact R&M.

h. Application Specific - DoD applications will serve as conduits for the transition of new technologies into "target-of-opportunity" military system programs. The products of the STARS program will be molded into DoD-specific applications. In particular, activities will address the technologies of reuseable components, component composition systems, very high level languages, application generators, and knowledge-based systems, each of which can have a significant impact on software R&M.

The STARS program addresses many aspects of software. Reliability is included in the planned activities but it is not addressed in an integrated manner. The principal proposed means of integrating efforts is to focus on development of a complete life cycle environment. Integrating tools into a coherent environment requires a conceptual basis for the integration. Any number of different conceptual or methodological bases for integration could and should be investigated in the STARS program. The study group feels that STARS should include the development of a reliability-based environment encompassing the aspects of software reliability discussed in this report. This effort will tend to focus technology development on reliability in a more integrated manner. The lessons learned from experimentation with such an environment can be factored into the more general environment to be created by STARS.

REFERENCES

1. Science, 22 Sept 78, "Technology Creep and the Arms Race: ICBM Problem a Sleeper."

2. Aviation Week and Space Technology, 19 July 1983, "Air Defense Missiles Limited Tactics of Argentine Aircraft."

3. Report of the DoD Joint Service Task Force on Software Problems, Department of Defense, 30 July 1982.

4. Software Technology for Adaptable, Reliable Systems (STARS) Program Management Plan, Department of Defense, 31 March 1983.

5. Avizienis, Algerdas, "Fault-Tolerance: The Survival Attribute of Digital Systems," Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

6. Champine, George A. "What Makes a System Reliable?" Datamation, September 1978.

7. Christian, Flavice "Exception Handling and Software Fault Tolerance", IEEE Transaction on Computers, Vol. C-31, No. 6 June 1982.

8. Hopkins, Albert L., Smith, T. Basil, and Lala, Jaynarahan H. "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft" Proceeding of the IEEE, Vol. 66, No. 10, October 1978.

9. Ossfeldt, Bengt E. and Jonsson, Ingmar, "Recovery and Diagnostics in the Central Control of the AXE Switching System" IEEE Transactions on Computers, Vol. C-29, No. 6 June 1980.

10. Quantitative Software Models, Data and Analysis Center for Software, RADC/ISISI, Griffis AFB, NY 13441, Report #SRR-1, March 1979.

11. "Special Issue on Software Reliability," IEEE Transactions on Reliability, Vol. R-28, No. 3 August 1979.

12. Lehman, M. and Belady, L., "Programming System Dynamics", ACM SIGOPS Third Symposium on Operating System Principles, October 1971.

13. Mills, H.D. "On the Statistical Validation of Computer Programs." 1970, Software Productivity, Little, Brown Computer Systems Series, 1983.

14. Curtis B., et al. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics, "IEEE Transactions on Software Engineering," March 1979.

15. Curtis, B. "In Search of Software Complexity," Workshop on Quantitative Software Models, IEEE Catalog No. TH0067-9, October 1979.

16. Proceedings of the National Conference on Software Test and Evaluation, National Security Industrial Association, Washington, DC, February 1983.

17. Koch, H.S. and Kubat, P., "Optimal Release Time of Computer Software," IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, May 1983.

18. Anderson T. and Knight J.C. "A Framework for Software Fault Tolerance in Real-Time Systems", IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, May 1983.

END

FILMED

3-84

DTIC